# A Pre-processing Tool for Z2SAL to Broaden Support for Model Checking Z Specifications

Maria Ulfah Siregar[1,2]

[1] PhD Student in the Department of Computer Science, The University of Sheffield
Regent Court, 211 Portobello, Sheffield, S1 4DP, United Kingdom
`acp12mus@sheffield.ac.uk`
[2] Lecturer in Informatics Department, UIN Sunan Kalijaga
Yogyakarta, Indonesia

**Abstract.** One of the deficiencies of Z tools is that there is limited support for model checking Z specifications. Building a model checker directly for a Z specification will take considerable amount of effort and time due to the abstraction of the language. Translating a Z specification input into a specification in a language that an existing model checker tool accepts is an alternative method. Researchers at the University of Sheffield implemented a translation tool, Z2SAL, that takes a Z specification and translates it into the input for Symbolic Analysis Laboratory (SAL), a framework for combining different tools for abstraction, program analysis, theorem proving and model checking. This paper discusses support for model checking Z specifications, in which the capability of Z2SAL is extended. This support includes a translation of a generic constant and a schema calculus definition. Instead of translating these aspects of the Z language into the SAL language as Z2SAL does, a Z specification containing these two notations will be pre-processed, in which a generic constant definition is redefined to an equivalent axiomatic definition and a schema calculus definition is expanded to a new schema definition. As a result of a successful redefinition or expansion, a redefined or expanded Z specification is generated, otherwise the original Z specification is returned. Results show that the large number of our examples can be run successfully by our system. The redefined or expanded Z specification can be translated later by Z2SAL and the generated SAL file can be model checked or simulated by the SAL tool. Results also show that Z2SAL can translate outputs of our system to some extent. The majority of generated SAL files can be run by the SAL tool.

Keywords: Z, specification, generic constant, schema calculus, Z2SAL, SAL.

## 1 Introduction

As a formal language, the use of Z in academia and industry has increased considerably. This is because Z has been used successfully to address a large variety of problems and the international standard was also designed for this language. The use of Z can make a specification more formal and free from ambiguity. In addition, Z allows a specification to be analysed mechanically [1]. Designing a specification of a system enables a user to verify the system at an early stage of development. Early verification could avoid high cost of system implementation and test phases, if the specification was designed correctly [2–4]. Therefore, a specification is crucial for a system, especially if the system relates to the safety of property and/or life.

However, there is a lack of tools for this language, especially in model checking Z specifications. Although the Community Z Tools (CZT) project is developing a set of open source tools for Z, progress of this development is slow [5]. There are many causes of the shortage of Z tools. These are mostly related to the Z language and semantics, such as an inherent expressiveness and a difficulty in deciding effectively any theorem about Z specifications [5, 1]. Another cause is the richness of this language, which can also be the issue of verifying Z [1]. Furthermore, only a few of these tools can be used in validating intended meanings of such Z specifications [6].

The lack of supporting tools for the Z language and the above issues has led researchers suggest alternative methods, which is a more rapid approach, to address this problem: reuse and adapt existing tools. Researchers at the University of Sheffield implemented the Z2SAL translator [5] which generates a SAL specification from a Z specification input. The generated SAL file can be model checked later by the SAL model checker. A brief introduction to Z2SAL and SAL is given in Section 2.

In our study, several experiments using Z2SAL and SAL are performed. Our finding is Z2SAL supports many Z tags, but not all. Furthermore, sometimes several generated SAL files cannot be verified or simulated by the SAL tool.

Therefore, this paper intends to address problems as stated below:

1. What are crucial features of Z should be implemented to enhance the ability of Z2SAL and why?
2. How to implement such features that are supported by Z2SAL and SAL?

These questions will be explored in the following sub-sections. Both the below sub-sections did not exist in [12].

### 1.1 Motivation

Based on our experiences using Z2SAL, two aspects of the Z notation were chosen to study. Both aspects will be discussed in this section.

The first aspect is the Z generic construct. Z2SAL cannot translate specifications that consist of generic constructs. As a result, error files were generated instead. Our finding is that Z2SAL cannot recognize a generic constant which is one type of the Z generic constructs. Although it has been declared in the generic constant definition, Z2SAL reported that the generic constant is a new identifier.

Z2SAL has not encountered any generic construct on Z specifications beforehand, so this part of Z has not been implemented yet. Therefore, our assumption is that the current version of Z2SAL does not support translation of either a generic constant or a generic schema. Although, Z2SAL's researchers might implement them.

Our study in the SAL literature concludes that a generic form cannot be found in the SAL language. Thus, another assumption is that Z2SAL does not support generic constructs in order to be consistent with the SAL language.

Specified using generic parameters, a generic constant is commonly used in formulating mathematical tool-kit operators [7], in which the operators do not depend on the particular type of elements in their construction [8]. Another usage of a generic constant is to specify a general notion which is used frequently in a system.

In case there is no generic constant, several equivalent functions should be formulated because each function is dedicated to one set of types of parameters; it is redundant work. Thus, a generic constant is beneficial to a Z specification.

The second aspect is the Z schema calculus. Z2SAL supports a translation of several schema calculus such as a schema inclusion, the $\Delta$ operator, and the $\Xi$ operator. However, they must be specified either vertically or horizontally in a schema. If a new schema is specified as being constructed from earlier schemas, Z2SAL does not support this schema construction. Thus, it is argued that Z2SAL does not support schema calculus.

The constructed schema is used commonly to define a more complex, modular and larger specification of a system. Therefore, schemas are reused to create new schemas. These schemas are combined by using schema operators. Different schema operators which are used define different new schemas.

Therefore, focus was set on generic constants and schema calculus as crucial features of the Z notation in our work. They were studied to extend Z2SAL so it can translate both of them. These findings were used to define our objective as discussed below.

## 1.2 Objective and Contribution

Our objective is to implement a tool. This tool will redefine a generic constant definition to an equivalent axiomatic definition based on usages of this generic constant.

Another objective is to implement a tool to create a new schema by expanding other schemas. These schemas are connected by schema operators.

Both these tools are implemented in a system which is called support for model checking Z specifications. This System is our contribution to broaden the applicability of model checking Z specifications. JFlex [9], BYACC/J [10], and Java language [11] were used to implement our system. During our experiments with this system, there is another contribution of a SAL translation of user defined functions and constants. It will be discussed later.

The paper is organized as follows. Section 2 describes briefly an introduction to Z2SAL and SAL. Section 3 contains our support for model checking Z specifications. This section also discusses how to implement this support which is supported by both Z2SAL and SAL. It is divided into two sub-sections. Each sub-section has been extended from its previous version in [12]. Section 3.1 outlines our support for generic constants. There are several new sub-sections in this section, such as Generic Abbreviation Definitions, Lambda Expressions, Summaries of Experiments on the Redefinition System, Size of Z Specifications, and Manual Modification in SAL files. However, majority of the contents of the first three ones and the last one have been discussed in [12]. The new sub-section, Size of Z Specifications, discusses how our redefinition system scales to a variety of sizes of Z specifications. Section 3.2 explains another support which is schema calculus. There is a new sub-section in this section, Summaries of Experiments with the Expansion System. Several contents of this sub-section were gathered from earlier sub-section. There are several new experiments have been performed in the expansion system as compared to the previous version in [12]. These experiments were summarised in new tables accompanied the new sub-section. Section 4 concludes this paper and summarises future work. This last section has also been extended from its previous version in [12].

## 2 A Brief Introduction to Z2SAL and SAL

Several tools in Z have been developed based on the quick approach, such as *ProZ* [6] which is a translator of Z into the existing Alloy Analyser tool, *ProB* [13]; data refinement verification [14] which uses Alloy SAT-solver based on a counter-example finder; and Z2SAL [5] which is a translator of a Z language specification into a SAL language specification [15].

Smith and Wildman at the University of Queensland, Australia, described how to translate a Z language specification into a SAL input language specification [16]. This basic idea was implemented in a tool set [17] and the current Z2SAL extends it in a different direction, to tackle optimization issues [5].

In providing a translator of Z into an input language of existing tools, SAL was chosen since it has an *equivalent representation* of many aspects of Z [17]. Moreover, 'many different tools exist, which use the SAL input language such as simulator, model checker either symbolic or bounded, deadlock checker, etc' [5], which are offered freely by Stanford Research Institute (SRI) International under academic licences.

A generated SAL file consists of a SAL module and/ or several SAL contexts. This module describes a transition system of Z states [17]. The simple SAL module has a general format as follows:

```
State : MODULE =
  BEGIN
    INPUT ...
    LOCAL ...
    OUTPUT ...
    INITIALIZATION [ ... ]
    TRANSITION [
       ...
    ]
  END;
```

The SAL context is a place to declare types, constants, modules and modules properties [15]. Z2SAL formulates several Z mathematical tool-kits, which are necessary for a generated SAL specification, in separate but integrated SAL context files.

Translating a Z language specification into a SAL input language specification requires several adjustments due to a number of differences of both languages [5]. These adjustments are discussed briefly as given below:

First, is *bounding the infinite*. Z supports *fully abstract* (non-grounded, non-constructive) specification styles, whereas SAL is a *concrete and grounded language*. For example, Z supports the built-in numerical types "$\mathbb{Z}$", "$\mathbb{N}$" and "$\mathbb{N}_1$", whose ranges are infinite. On the other hand, SAL has similar unbounded types `INTEGER`, `NATURAL` and `NZNATURAL`, which can be used only as base types of finite sub-ranges in a SAL specification. Z also supports given types, which have semantics of an un-interpreted set, such as `[TAPE, NAME]`. Therefore, the translations provided by Z2SAL should specify a finite number for sizes of these sets.

The *mismatched formal paradigms* are the second difference. Z and SAL have very different styles of specifications and descriptions. The Z specification, which consists of state schemas and operational schemas, is built-up increasingly. It views locally and functionally such that every operational schema operates on its input and output variables, or on variables of state schemas. On the other hand, the SAL specification is created as a 'monolithic finite state automaton' (FSA) such that all inputs, outputs and local variables are compiled into aggregate states [5]. Moreover, all operations act upon guard transitions from one state configuration to another state configuration [5]. Thus, this mismatch can be approached by re-ordering all information in the Z specification. A further mismatch is that Z specifications often use partial functions [5]. On the other hand, as SAL is based on *Binary Decision Diagrams* (BDDs), SAL always requires a representation of function as a total function. Thus, a work-around is necessary in order to present a partial function in Z specifications as a total function in SAL. Furthermore, a set cannot be treated as a monolithic FSA of SAL, but as a 'poly-lithic collection of judgements' over its elements instead [5]. Thus, several operations in sets are necessary to be expressed differently, such as the cardinality of a set, which is not supported by SAL.

The last difference is an issue of *non-computable specifications* [5]. A Z specification naturally supports non-constructive styles of a specification. These styles should be expressed in computable styles of a specification in SAL. Both styles essentially are different indeed. Normally, a SAL specification consists of a set of update assignments to primed variables, which indicates posterior variable states. On the other hand, a direction of a constructive approach is not necessary in a Z specification. Z2SAL asserts posterior existences of variables and restricts their values on preconditions. This requires a search for suitable precondition values.

More information relating to Z2SAL is provided in [18]. It also includes a downloadable version of this translation tool.

SAL is a framework of several different tools such as abstraction, program analysis, theorem proving and model checking, which is used to change concepts and implementations of model checkers and theorem provers. These concepts and implementations at first were based on verification, but they were extended to include calculation of properties or symbolic analysis such as abstraction, slicing and composition [15, 19].

The SAL language can be used as a specification language, a target language for several translators, or a common source of several analysis tools. It is originated of a collaboration of two researchers, David Dill from Stanford University and Thomas Henzinger from the University of California at Berkeley. These collaborations devolved SAL further and incorporated Verimag. SAL is now developed at SRI and its current version is 3.3. The SAL language syntax can be found in [15].

The next section describes our support for model checking Z specifications.

## 3 Support for Model Checking Z Specifications

As mentioned earlier in Section 1.2, there are two main types of our support for model checking Z specifications. The first is a generic constant, which will be described in the following sub-section.

### 3.1 Support for Generic Constants

Our first support is to aid Z2SAL to translate generic constants. The following sub-sections describe briefly an introduction to a generic constant and our system, also discuss results of several examples.

**Introduction** A generic constant is used to introduce a new constant which uses generic parameters [7]. By using a generic parameter, different types of a parameter can be specified. They are specified by using different literals such as `X`, `Y`, `Z` and others. A generic constant has a global scope in a Z specification, whereas a generic parameter has a local scope in the particular generic constant definition.

An example of a generic constant definition is formulated as follows:

$$\begin{array}{|l}
[X] \\
\hline
monoSequence : \mathbb{P}(\mathrm{seq}\, X) \\
\hline
monoSequence = \{ s : \mathrm{seq}\, X \mid \#(\mathrm{ran}\, s) \leq 1 \} \\
\hline
\end{array}$$

The above definition has `monoSequence` as the generic constant, which is a constant (see a discussion below). The output type is a set of a sequence of `X`. There is one specified generic parameter, `X`. This generic constant definition defines a set of a sequence of `s`, which just has at the most one element.

**A Generic Constants Redefinition System** Our approach to support Z2SAL in translating generic constant definitions is to implement a tool. This tool will redefine a generic constant definition to an equivalent axiomatic definition based on usages of this generic constant (see Section 1.2). This approach is based on similar behaviour between a generic constant and an axiomatic definition. In other words, they both declare a global variable inside a Z specification. This redefinition is called an actualization process, in which a generic typed parameter will be actualised to its actual typed parameter.

Plagge and Leuschel in [13] also proposed the same method as our method for translating a generic definition defined in a Z specification. As discussed in their paper, generic constant definitions had not been added to Z specification examples.

Our system specifies different types of generic constants. These types can be identified based on the generic constant declarations, as given below:

– a function; the outermost operator is an infix generic function. A complete set of these functions is "$\pfun$", "$\rightarrow$", "$\pinj$", "$\rightarrowtail$", "$\psurj$", "$\twoheadrightarrow$" and "$\bij$". These functions are collected in one token, `INGEN`. As a function, it will have at least one input parameter and one output parameter. This type can be generic.
– a relation; a declaration uses the "$\leftrightarrow$" tag in its outermost operator. This tag has the `REL` string as its token. As a relation, there is no output parameter type. In other words, the output is the relation itself; a pair of types.
– a constant; a constant means it does not require any input. Thus, a declaration of this generic constant only gives us generic output parameters. This declaration denotes none of the above tags in its outermost declaration.

The above three types of generic constants are parts of a variable declaration of the Z grammar in the Z language. This grammar, which refers to [8], was specified in our parser as follows:

```
expr1:  expr1.word REL decor expr1.word
     |  expr1.word INGEN decor expr1.word
     |  expr2.chain
     |  expr2
     ;
```

The first production rule indicates a relation, whereas the second one is a function. The third production rule contains `CROSS` obtained from `expr2.chain`. Thus, this production rule can either be a function or a relation depending on which of those first two production rules is fired previously. The last one is a constant; both function and relation production rules are not matched.

Inevitably, a constant actualization is not always straightforward, especially a constant implicit type. In this case, a solution is to infer the actual type of the generic constant.

Our redefinition system is intended as a pre-processing tool which can aid Z2SAL. A Z specification input, which consists of generic constant definitions and usages, will be pre-processed by this tool in order to redefine its generic constant definitions.

This tool was implemented in Java. It has a simple GUI to interact with users and has also two preliminary processes: the scanner and the parser generation. These two generators were implemented by using the JFlex scanner generator [9] and the BYACC/J parser generator [10] respectively.

The current version of our system implemented several Z tokens which refer to [8, 20] and several production rules of the Z grammar which refer to [8]. Our system also experienced of simple variable types of generic constants.

The next sub-section discusses an example of the redefinition process. This Z specification was taken from [21], **the swap function**.

**An Example of the Redefinition Process** This specification has one given type, `NAME`. There are two generic constant definitions for the swap process defined in this specification. These functions, each of which has two parameters, swap the orders of its parameters. Thus, after a swap, an element in the second position will be shifted such that this element is in the first position and vice versa.

The first definition, as shown below, has two different generic parameters: `X` and `Y`. These different parameters mean that both of them have different types. The generic constant is `swap2` which is shown in the following example:

$$[X, Y]$$
$$swap2 : X \times Y \rightarrow Y \times X$$
$$\forall\, x : X;\ y : Y \bullet swap2(x, y) = (y, x)$$

The second definition has one generic parameter, `X`. This single parameter means that the swap process will occur on objects of the same type. The generic constant is `swap1` which is shown in the following example:

$$[X]$$
$$swap1 : X \times X \rightarrow X \times X$$
$$\forall\, x, y : X \bullet swap1(x, y) = (y, x)$$

A state schema, `State`, has only one state variable, `name`, which is an instance of the specified given type. There is no predicate specified in this schema.

The initialization schema, `Init`, refers to the post state of the state schema. This schema does not declare its own variable and predicate. It means that this schema only contains predicates which are inherited from its reference to the state schema. In this case, the reference is the post state of the state schema.

There is one operational schema specified in this specification, `Swap`, which calls these generic constants. This schema does not change a state of the system indicated by a reference to $\Xi$ `State`. This schema is specified as follows:

$$Swap$$
$$a? : NAME;\ a!, b! : NAME;\ c? : \mathbb{N};\ c! : \mathbb{N};\ \Xi State$$
$$(b!, a!) = swap1[NAME, NAME](name, a?)$$
$$(c!, a!) = swap2(name, c?)$$

As can be seen in the above schema, each generic constant has one usage. The first usage uses explicit parameter types in addition to parameters required by the function. Our system generates two axiomatic definitions for these usages as shown below:

$$\begin{array}{|l} swap1 : NAME \times NAME \rightarrow NAME \times NAME \\ \hline \forall\, x, y : NAME \bullet swap1(x, y) = (y, x) \end{array}$$

$$\begin{array}{|l} swap2 : NAME \times \mathbb{N} \rightarrow \mathbb{N} \times NAME \\ \hline \forall\, x : NAME;\ y : \mathbb{N} \bullet swap2(x, y) = (y, x) \end{array}$$

Consider that the explicit type has been deleted from the first usage since Z2SAL does not support this type of parameter. Thus, the first usage should be modified by our system as follows:

$$(b!, a!) = swap1(name, a?)$$

This modification was conducted on this usage to let Z2SAL translates this specification successfully.

**Result and Discussion** The generated specification of the above example can be translated by Z2SAL. A SAL file, generated by Z2SAL, can also be verified by the SAL model checker. However, it failed to be simulated by the SAL model checker. This simulator generated an unsupported error of a failure to convert function application.

Furthermore, if a theorem was added to the generated SAL file, this SAL file cannot be verified either by the SAL model checker. Thus, it is an issue of the redefinition system.

The current Z2SAL translates the Z functions, relations and constants, and puts them in the base module. Z2SAL defines `State` as the default name for this module. The simple structure of this module can be seen in Section 2. This translator also puts variable declarations in a definition clause. The definition clause is part of the base module or in other words it is inside the base module.

As a result, an error was sometimes experienced during our experiments with user-defined functions. This error related to an incompatible type in the equality operator or a failure to convert function application produced by the SAL model checker or simulator, as given earlier.

A user defined function, relation and constant are always declared outside a SAL module [15]. They are put in a context clause, specifically in a constant declaration, instead. The module language in SAL describes transition system modules [15]. However, it cannot be used to declare new types or constants or asserting properties of the module [15]. All of these can be easily declared by specifying them in the SAL context language.

A translation method of user defined functions adapted by the SAL language is different to the one that Z2SAL adapts. Based on this finding, the same method as SAL's method was proposed by us to Z2SAL researchers during our study. This proposal can be considered as our contribution in model checking Z specification as mentioned in Section 1.2.

This method can be applied to a user defined function and constant, but it is not applicable to a user defined relation since a relation does not have a type for its output parameter. It is based on a signature of this SAL function specified in [15].

The signature of which is named as a constant declaration has the following syntax rule[15]:

$$\mathit{ConstantDeclaration} := \mathit{Identifier}[(\mathit{VarDecls})] : \mathit{Type}[= \mathit{Expression}]$$

This constant declaration, as mentioned above, is part of the SAL context language. The SAL context language syntax is given as follows [15]:

| | | |
|---|---|---|
| *Context* | ::= | $\mathit{Identifier}[\{\mathit{Parameters}\}] : \mathit{CONTEXT} = \mathit{ContextBody}$ |
| *Parameters* | ::= | $[\mathit{TypeDecls}];\ \{\mathit{VarDecls}\}^{*,}$ |
| *TypeDecls* | ::= | $\{\mathit{Identifier}\}^{+,} : \mathit{TYPE}$ |
| *ContextBody* | ::= | *BEGINDeclarationsEND* |
| *Declarations* | ::= | *ConstantDeclaration* |
| | \| | *TypeDeclaration* |
| | \| | *AssertionDeclaration* |
| | \| | *ContextDeclaration* |
| | \| | *ModuleDeclaration* |
| *ConstantDeclaration* | ::= | $\mathit{Identifier}[(\mathit{VarDecls})] : \mathit{Type}[= \mathit{Expression}]$ |
| *TypeDeclaration* | ::= | $\mathit{Identifier} : \mathit{TYPE}[= \mathit{TypeDef}]$ |
| *AssertionDeclaration* | ::= | $\mathit{Identifier} : \mathit{AssertionForm} = \mathit{AssertionExpression}$ |
| *AssertionForm* | ::= | $\mathit{OBLIGATION} \mid \mathit{CLAIM} \mid \mathit{LEMMA} \mid \mathit{THEOREM}$ |
| *ContextDeclaration* | ::= | $\mathit{Identifier} : \mathit{CONTEXT} = \mathit{Identifier}\{\mathit{ActualParameters}\}$ |
| *ActualParameters* | ::= | $\{\mathit{Type}\}^{*,};\ \{\mathit{Expression}\}^{*,}$ |

Other non-terminals or rules can be found in the same reference as given above.

Thus, the generated SAL file was modified to adapt a constant declaration formulated by SAL. Both the above function definitions were formulated manually on the generated SAL file. They are shown below:

```
swap1(q__1 : NAME, q__2 : NAME): B__NAME__X__B__NAME = (q__2,q__1);

swap2(q__3 : NAME, q__4 : NAT): B__NAT__X__B__NAME = (q__4,q__3);
```

Original declarations generated by Z2SAL for these functions were deleted.

A few theorems were added to this specification as shown below:

```
th1: theorem State |- G(FORALL (i: NAME,j: NAT): swap2(i,j) = (j,i));

th2: theorem State |- G(FORALL (i,j: NAME): i = j =>
                          swap1(i,j) = swap1(j,i));

th3: theorem State |- G(FORALL (i,j: NAME): swap1(i,j) = swap1(j,i));
```

The first two theorems are valid; the swap system can satisfy both properties. The last theorem is invalid since the swap function will not give us the same result for different parameters.

There is another issue relating to an abbreviation definition and a lambda expression which was found during our experiments with the redefinition system. Both these issues will be discussed in the next sub-sections.

**Generic Abbreviation Definitions** Z2SAL supports an abbreviation definition, but not the generic one. Declaring a global constant by using an abbreviation definition is common in writing Z specifications. Thus, a generic abbreviation definition was taken also into our consideration.

In the case of generic abbreviations, it is not enough just to work with an actualization of a generic type. The other issue here is a set comprehension definition. A generic abbreviation definition is usually defined by using a set comprehension definition. However, Z2SAL does not support an abbreviation definition consisting of a set comprehension.

For example, consider a generic abbreviation definition as below [7]:

$$monoSequence[X] == \{s : \mathrm{seq}\, X \mid \#(\mathrm{ran}\, s) \leq 1\}$$

A generic abbreviation definition can be rewritten to a generic constant definition. Both these definitions declare global constants in the related Z specification, in this case the type of the generic constant is a constant.

The expression in the right hand side of the "==" uses a set comprehension definition, which denotes that `monoSequence` is a set of a sequence of `X`. The body of this generic definition is obtained from the expression after the "==" tag.

Thus, a generic abbreviation definition is first rewritten to a generic constant definition. This rewriting is performed manually and automatically in order to prove that it is correct. This equivalent definition was given in Section 3.1. Afterwards, this generic constant definition is redefined to an axiomatic definition.

**Lambda Expressions** Another kind of generic forms is the "$\lambda$" expression, which is used to define a function without specifying a name [7]. Z2SAL does not support this expression which is common in generic constant definitions or in other definitions in a Z specification generally. Our approach is to rewrite a lambda expression automatically and manually to an equivalent expression without any lambda expression. Then, it is redefined to an axiomatic definition.

For example, a generic constant definition as formulated below consists of the lambda expression [7]:

$$\begin{array}{|l}
[X] \\
\hline
commonSubseq : ((\mathrm{seq}\, X) \times (\mathrm{seq}\, X)) \rightarrow \mathbb{P}(\mathrm{seq}\, X) \\
\hline
commonSubseq = (\lambda\, s, t : \mathrm{seq}\, X \bullet allSubseqs \cap allSubseqt) \\
\hline
\end{array}$$

The lambda expression in the above definition can be rewritten to an equivalent definition as follows:

$$commonSubseq = \{s, t : \mathrm{seq}\, X \bullet ((s, t), allSubseq(s) \cap allSubseq(t))\}$$

or another equivalent one as given below:

$$\forall\, s, t : \mathrm{seq}\, X \bullet commonSubseq(s, t) = allSubseqs \cap allSubseqt$$

A lambda expression definition, ($\lambda$ `S` $\bullet$ `E`), represents a function and has arguments which are taken from `S`. An output of this expression is the value of `E` [8].

As given by the first equivalent definition above, the lambda expression is equivalent to a set comprehension, $\{S \bullet (T, E)\}$, in which `T` is a characteristic tuple of `S`. In a set comprehension, a characteristic tuple is obtained from its declaration. Thus, `(s,t)` is the characteristic tuple of the above set comprehension.

During our experiments, Z2SAL was unable to translate a set comprehension definition with many parameters of the same type. According to the SAL grammar rules, only one parameter can be declared in one definition of a set comprehension. The SAL syntax [15] for a set expression is given as follows:

$$\begin{aligned}&\mathit{SetExpression} := \mathit{SetListExpression} \mid \mathit{SetPredExpression}\\&\mathit{SetListExpression} := \{\{\mathit{Expression}\}^{+}_{,}\}\\&\mathit{SetPredExpression} := \{\mathit{Identifier} : \mathit{Type} = \mathit{Expression}\}\end{aligned}$$

Thus, our approach is to rewrite the first equivalent lambda expression to the second equivalent one.

Several results collected from our experiments are summarized and discussed in the next sub-section.

**Summaries of Experiments on the Redefinition System** A number of experiments on several Z specifications are presented on Table 1. These experiments run on a laptop with a 1.30GHz Genuine Intel(R) CPU U7300 and 2.00 GB RAM.

The second column of Table 1 indicates that a manual modification was made to the SAL file. The SAL file was generated by Z2SAL from the Z specification produced by our redefinition system. This modification is required so that the SAL file can be verified by the SAL model checker or simulated by the SAL simulator. It involved rewriting a user defined function and placing this function in which SAL put its function. Examples of this rewriting were given earlier in Section 3.1. The modification also involved rewriting other parts of a SAL file. Such a modification implies that there is a bug in the translation of associated Z specification by Z2SAL. It can also be a mismatch between the Z language and the SAL language. This manual modification will be discussed in the later sub-section.

The third column shows verification times of each SAL file. A SAL file which has one verification time means that this file has only be verified for one case of the number of theorems. A SAL file which has two verification times means that the SAL file at first can be verified by the SAL model checker. However, later it cannot be verified if at least one theorem was added to this SAL file. Such a SAL file usually cannot be simulated either by the SAL simulator even there is no theorem.

Based on our experiments as shown in Table 1, majority of SAL files generated by Z2SAL from the output of our system, can be verified by the SAL model checker. It is proved by existences of a verification time in each row of the table.

Inevitably, there is one output produced by our system which cannot be translated by Z2SAL. The output is generated from the **fFileStorage.tex** input file. It is because this Z specification contains a function which its range is also

**Table 1.** Several Experiments with the Redefinition System

| **Z Specification** | **Details** | **Verification time in secs** | |
|---|---|---|---|
| **(*.tex)** | | **#Theorem = 0** | **#Theorem > 0** |
| bbook | Modified SAL function | | 0.842 |
| bbook_map | Modified SAL function | 0.016 | 0.25 |
| bbook_uni | Modified SAL function and other parts of SAL file | 0.031 | 0.406 |
| bbook_map_uni | Modified SAL function and other parts of SAL file | | 0.359 |
| fDomRan | Modified SAL function | 0.015 | |
| fEmpty | OK | | 0.093 |
| fEmptyImpl | OK | | 0.109 |
| fFirst | Modified SAL function | 0.015 | 0.187 |
| fHead | Modified SAL function | 0.031 | |
| fHeadFunc | Modified SAL function and cannot be simulated: The set of initial states is empty | 0.031 | |
| fMaxComSubSeq | Modified other parts of SAL file and cannot be simula ted: An out of memory error | 0.047 | |
| fMaxComSubSeq_1 | Modified other parts of SAL file and cannot be simula ted: An out of memory error | 0.032 | |
| fMaxComSubSeq_orig | Modified other parts of SAL file and cannot be simula ted: An out of memory error | 0.032 | |
| fMonoSeq | OK. Long simulation | 0.047 | |
| fMonoSeq_1 | OK. Long simulation | 0.031 | |
| fSwap | Modified SAL function | 0.016 | 0.141 |
| fUniqSeq | Ok. Cannot be simulated: An out of memory error | 0.062 | |
| fUniq1Seq | Ok. Cannot be simulated: An out of memory error | 0.031 | |
| fUniq2Seq | Ok. Cannot be simulated: An out of memory error | 0.015 | |
| tn | Modified other parts of SAL file and cannot be simula ted: An out of memory error | 0.03 | |
| tnImpl | Modified other parts of SAL file and cannot be simula ted: An out of memory error | 0.0 | |
| fFileStorage | Canot be translated by Z2SAL | N/A | |
| fSet | Modified SAL function and other parts of SAL file | 0.0 | |

a function. Z2SAL does not support such a type. A quick solution is to rewrite such a function. However, another error relating to the "`..`" tag was experienced. It was concluded that it is a bug in a Z2SAL translation of a range of numbers.

Another finding is that a few SAL files, which can be verified by the SAL model checker, cannot be simulated by the SAL simulator due to an out of memory error as can be seen in Table 1. These SAL files usually have sequences or a set inside other sets. Currently, this error has not been solved.

Relating to out of memory errors, the following sub-section discusses this issue. The discussion will be accompanied by a table.

**Size of Z Specifications** Z specifications used for our experiments have different sizes measured in kilobytes. These sizes are summarized in Table 2. Sizes of the redefined specifications are recorded also on this table.

"`input`" on this table means a Z specification input file for our system. On the other hand, "`output`" means a Z specification output file generated by our system after performing a redefinition process, "N/A"s in `SAL specifications` means an associated Z specification cannot be translated by Z2SAL.

Referring to this table, almost all of our experiments have the same sizes of Z specifications before and after redefinition processes. It means that there were not many usages specified in these specifications. It can also mean that the generic constant definitions are not quite complex definitions.

The size of a SAL specification is roughly twice to four times of its Z specification. Sizes of SAL specifications shown in this table are original sizes producing by Z2SAL. As discussed above, several of these SAL specifications have been modified as required in order to be executed by the SAL tool successfully. Thus, their sizes can be different from the original ones.

There are only four experiments which their sizes of Z specification outputs were increased twice of their inputs. These specifications are `bbook_map_uni`, `fUniqSeq`, `fUniq1Seq` and `fUniq2Seq`.

As shown in Table 1, several of our Z specifications cannot be simulated by the SAL simulator because of out of memory errors. However, these errors cannot be blamed for the increasing sizes of specifications. It is because there are other specifications which their sizes were not increased, but they were involved on the same errors as above. Sizes of these specifications are greater than 1. However, it can coincide which is not influenced entirely only by sizes of specifications.

An additional factor is the complexity of a declaration of a generic constant. The out of memory errors were involved on specifications which have either sequences, or sets of sets.

Thus, a Z specification, which does not have a sequence, a set of other set, or a range of numbers, can be executed successfully by the SAL tool. It argues also that a size of a generic constant definition and a number of usages relate to a generation of that error.

In a conclusion, our approach to redefine generic constants definitions scales to larger specifications. However, as the outcomes of our system will be translated

**Table 2.** Sizes of Z Specifications

| **Z Specifications** | **Sizes in KB** | | |
|---|---|---|---|
| **(.tex)** | **input** | **output** | **SAL specifications** |
| bbook | 2 | 2 | 6 |
| bbook_map | 1 | 1 | 4 |
| bbook_uni | 1 | 1 | 4 |
| bbook_map_uni | 1 | 2 | 5 |
| fDomRan | 2 | 2 | 6 |
| fEmpty | 1 | 1 | 2 |
| fEmptyImpl | 1 | 1 | 2 |
| fFirst | 1 | 1 | 3 |
| fHead | 1 | 1 | 3 |
| fHeadFunc | 1 | 1 | 3 |
| fMaxComSubSeq | 2 | 2 | 4 |
| fMaxComSubSeq_1 | 2 | 2 | 4 |
| fMaxComSubSeq_orig | 2 | 2 | 4 |
| fMonoSeq | 1 | 1 | 3 |
| fMonoSeq_1 | 1 | 1 | 3 |
| fSwap | 1 | 1 | 2 |
| fUniqSeq | 1 | 2 | 5 |
| fUniq1Seq | 1 | 2 | 5 |
| fUniq2Seq | 1 | 2 | 5 |
| tn | 3 | 3 | 6 |
| tnImpl | 3 | 3 | 6 |
| fFileStorage | 2 | 2 | N/A |
| fSet | 2 | 2 | 5 |

by Z2SAL and executed by the SAL tool later, the large specification generated by our system is possible to be a problem with both tools.

As mentioned earlier, a separate discussion in manual modification in SAL files will be offered. The following sub-section discusses our approach to this manual modification.

**Manual Modification in SAL Files** Although all generated SAL files in our experiments with this system can be verified by the SAL model checker, a few of them at first failed. The modified version of these SAL files also failed to be verified by the SAL model checker. These files are **output_bbook_uni** as the SAL file generated from the output of **bbook_uni.tex**, **output_bbook_map_uni** as the SAL file generated from the output of **bbook_map_uni.tex** and **output_fSet** as the SAL file generated from the output of **fSet.tex**.

This failure related to incompatible types in the equality operator. The SAL model checker identified that the type of `birthday` is not compatible with the type of the first argument of a function `uniSet` in the first and second SAL files. The `uniSet` function which is a generic constant definition was specified as follows:

$$[X]$$

$$uniSet : (\mathbb{P}\, X) \times (\mathbb{P}\, X) \rightarrow (\mathbb{P}\, X)$$

$$\forall\, S, T : (\mathbb{P}\, X) \bullet uniSet(S, T) = \{x : X \mid x \in S \vee x \in T\}$$

This function combines two sets of elements which have the same types. As can be seen, this function requires two parameter inputs. Both of them have the same types as the output type.

An example of usage of the above generic constant is specified as follows:

$$birthday' = uniSet(birthday, \{name? \mapsto date?\})$$

As can be seen from the above generic constant definition, the type for the first parameter is a set of `X`. This type is an expected type. On the other hand, `birthday` is the first parameter passed to `uniSet`. The type of `birthday` will be the actual type for this parameter. The declaration of the function `birthday` is as follows:

$$birthday : NAME \nrightarrow DATE$$

`birthday` is a state variable, which is a partial function from `NAME` to `DATE`.

Our system generated the `uniSet` axiomatic definition as follows:

$$uniSet : (\mathbb{P}(NAME \times DATE)) \times (\mathbb{P}(NAME \times DATE)) \rightarrow (\mathbb{P}(NAME \times DATE))$$

$$\forall\, S, T : (\mathbb{P}(NAME \times DATE)) \bullet uniSet(S, T) = \{x : (NAME \times DATE) \mid x \in S \vee x \in T\}$$

As can been from the above definition, the type of `birthday` has been modified to its equivalent type. It is done so to ease the unification of the expected type, `X`, and the actual type, `NAME` $\nrightarrow$ `DATE`.

A function type can be rewritten to a relation type [8]. Several constraints should also be added to maintain that it was a function. Furthermore, a relation is equivalent to a set of a pair of types.

$$X \leftrightarrow Y \equiv \mathbb{P}(X \times Y)$$

Thus, SAL failed to recognize that `birthday` had an equivalent type to the first argument of the `uniSet` user-defined function. This error indicated that there was incompatible type between the output of `uniSet`, `Set__C__B__NAME__X__B__DATE__I`, and the right hand side of the equality operator, `[NAME__X__DATE -> bool]`. Afterwards, a sequence of modifications was performed to the associated SAL file lines.

The last error produced by the SAL model checker is as follows:

```
Error: [Context: output_bbook_uni_mod, line(62), column(29)]:
Type mismatch in the function application.
Expected type:
[set{output_bbook_uni_mod!NAME__X__DATE}!Set,
set{output_bbook_uni_mod!NAME__X__DATE}!Set]
Actual type:
[output_bbook_uni_mod!Set__C__NAME__X__B__DATE__I,
set{output_bbook_uni_mod!NAME__X__DATE}!Set]
```

The related SAL lines are as follows:

```
61 NOT set {NAME;} ! contains?(known, name?) AND
62 birthday' = uniSet((birthday, set {NAME_X_DATE;} !
63 singleton((name?, date?)))) AND
64 invariant__'
```

In line 62, the type of `uniSet` after modification is a pair of `set {NAME_X_DATE;} ! Set` and `set {NAME_X_DATE;} ! Set`. This type was not compatible with the actual type passed to `uniSet`, which was a pair of `Set__C__NAME__X__B__DATE__I` and `set {NAME_X_DATE;} ! Set`. The type `Set__C__NAME__X__B__DATE__I` is an alias for `[NAME -> B__DATE]`, specified by Z2SAL.

Although a function is special type of a relation and a relation is a set of a pair of types in the Z language, SAL did not conclude that both types of the first argument of `uniSet` are the same. Thus, this incompatible type was solved manually. This is because our tool has not been able to perform this modification automatically.

Our last modification defined the same alias for `birthday`, but this time the alias represents a relation, not a function any more. It is shown as follows:

```
Set__C__NAME__X__B__DATE__I : TYPE = set {NAME_X_DATE;} ! Set;.
```

This change affects the usage of `birthday`. It cannot any longer be used as a function.

```
function {NAME, B__DATE; DATE__BB} ! partial?(birthday) AND
```

As a result, the above line was deleted from the old SAL file.

```
known = relation {NAME, DATE;} ! domain(birthday) AND
```

Another result is the above line, which is a relation, replaced a line, which is a function, as follows:

```
known = function {NAME, B__DATE; DATE__BB} ! domain(birthday) AND
```

As well as a line as follows, which presents a usage of a function:

```
date' = birthday(name?) AND
```

was replaced by a line below, which presents a usage of a relation:

```
set {NAME_X_DATE;} ! contains? (birthday, (name?, date')) AND
```

Finally, the modified SAL file can be verified by the SAL model checker and simulated by the SAL simulator.

The same function was also a source of the error in the third SAL file, but this time its first actual parameter is `used`. A usage of this function in the associated Z specification is as follows:

$$used' = uniSet(used, n)$$

The `used` state variable is a set of "$\mathbb{N}_1$" and the `n` variable is an instance of "$\mathbb{N}_1$".

An axiomatic definition generated for the above usage is as follows:

$$\begin{array}{|l} uniSet : (\mathbb{P}\,\mathbb{N}_1) \times (\mathbb{P}\,\mathbb{N}_1) \rightarrow (\mathbb{P}\,\mathbb{N}_1) \\ \hline \forall S, T : \mathbb{P}\,\mathbb{N}_1 \bullet uniSet(S, T) = \{x : \mathbb{N}_1 \mid x \in S \lor x \in T\} \end{array}$$

After a similar modification as performed in both SAL files above, the modified SAL file can be verified and simulated by the SAL tool.

Another aspect of the Z notation in our study is the schema calculus. This aspect was taken as the second type of our support for model checking Z specifications.

## 3.2 Support for Schema Calculus

This sub-section discusses the addition of support for Z schema calculus to our tool. The sub-section begins with an introduction to schema calculus. It is followed by a brief introduction to our support for this Z notation and our experiments on this system.

**Introduction** Z2SAL supports a translation of several schema calculus such as a schema inclusion, the $\Delta$ and the $\Xi$ operator, but they must be specified either vertically or horizontally in a schema. However, if a new schema is constructed from earlier schemas, Z2SAL does not support this schema construction. Thus, it argues that Z2SAL does not support schema calculus definitions.

The constructed schema is specified by using "$\widehat{=}$". It is the same as the supported schema calculus. However, the constructed schema does not use "$[$" and "$]$" to surround its declaration of variables and predicates.

The constructed schema is used commonly to define a more complex, modular and larger specification of a system. Schemas that have been specified can be reused to specify a new schema. It is because every schema has its distinctive operation in a specification, called 'schema separation' [2].

**A Schema Calculus Expansion System** Our approach is to construct a new schema by expanding other schemas, in which they are connected by schema operators. This system was included in the support tool for model checking Z specifications, the same as the redefinition system.

Since every schema operator has its own definition, a schema operator affects how the expansion is performed. The expansion means that all unique variables of involved schemas are listed in the new schema. It also means that predicates from the involved schemas are added. These predicates are combined using specified schema operators.

There is a prerequisite for operating two schemas; the same or common variables should have the same type. Furthermore, in a case of the negation operator, normalisation is also required.

Normalisation is to define explicitly the constraint given by the declaration part of the related schema. This constraint is specified in the predicate part. Normalisation should be performed just before the negation. This process is applied also to other schema operators for the sake of easiness. Several normalisation rules were specified in our system as follows:

– Every "$\mathbb{N}$" or "$\mathbb{N}_1$" in a declaration part is rewritten to a type of "$\mathbb{Z}$".
– Every "seq" or "$\mathrm{seq}_1$" is changed to $\mathbb{P}(\mathbb{Z} \times newVal)$, $newVal$ is a type which comes after "seq" or "$\mathrm{seq}_1$". The previous rule is applied also to $newVal$.
– Every function is changed to a pair of its left hand side type and its right hand side one. Both the above rules are also applied to the type in the left and in the right.

In general, after each schema is expanded, variables and predicates will be collapsed to a reference of a state schema. This collapse benefits the new schema to get a more compact schema and to avoid re-declarations of state variables.

Our system can expand several schema operators such as conjunction "$\wedge$", disjunction "$\vee$", negation "$\neg$", implication "$\Rightarrow$", bi-implication "$\Leftrightarrow$", hiding "$\setminus$", renaming "$/$", composition "$\fatsemi$", universal quantifier "$\forall$" and existential quantifier "$\exists$". Our system can also perform a simple simplification over a predicate part.

The next sub-section describes an example from our experiments with this system.

**An Experiment with the Schema Calculus Expansion System** An example, **expandingschema_3.tex**, will be presented in this sub-section. This example was taken from [2], but has been modified in some places for our experiments.

This example represents a library system specification. It has a state and an initialization schema as follows:

*Library*

$stock : COPY \nrightarrow BOOK;\ issued : COPY \leftrightarrow READER$
$shelved : \mathbb{F}\, COPY;\ readers : \mathbb{F}\, READER$

$\forall\, x : COPY;\ y1, y2 : READER \bullet$
$(x \mapsto y1) \in issued \wedge (x \mapsto y2) \in issued \Rightarrow y1 = y2$
$shelved \cup \mathrm{dom}\, issued = \mathrm{dom}\, stock$
$shelved \cap \mathrm{dom}\, issued = \varnothing$
$\mathrm{ran}\, issued \subseteq readers$
$\forall\, r : readers \bullet \#(issued \rhd \{r\}) \leq maxloans$

*InitLibrary*

$Library'$

$shelved' = \varnothing$
$readers' = \varnothing$

As can be seen from the above state schema, this library system has four state variables:

– `stock` is a partial function from `COPY` to `BOOK`. It gives us information about what copies a book has.

- `issued` is a relation between a copy of a book and a reader. It gives us information about which copy of a book each reader has.
- `shelved` is finite set of `COPY`.
- `readers` is a finite set of `READERS`.

This system has also three given types: `COPY, BOOK, READER`.

There is one schema calculus definition specified in this specification, which uses the Z schema composition operator, "$\mathbin{;}$". It is shown as follows:

$$Donate \mathrel{\widehat{=}} EnterNewCopy \mathbin{;} RegisterReader.$$

This operator will combine the second schema with the first schema, in which the result of the first schema is an input for operating the second schema.

The schema composition consists of the number of operations taken from other schema operators. Renaming is the first operation to take place. Its processes begin with renaming the same state variables so that the primed ones in the first schema and non-primed ones in the second schema have the same name of variables. Afterwards, these renamed schemas are combined using a conjunction operator. The next process is to hide the common renamed variables in a declaration part of the new schema. It is followed by adding an existential quantification which binds these hidden variables in a predicate part of the new schema.

The new schema, `Donate`, was constructed by our system as given below:

*Donate*

$\Delta Library;\ b? : BOOK;\ r? : READER;\ rep! : Report$

$\exists\, c : COPY \mid c \notin \operatorname{dom} stock \bullet (stock' = stock \oplus \{c \mapsto b?\} \land$
$shelved' = shelved \cup \{c\}) \land r? \notin readers \Rightarrow$
$(readers' = readers \cup \{r?\} \land rep! = Ok) \land$
$r? \in readers \Rightarrow (readers' = readers \land rep! = ReaderAlreadyRegistered)$
$\land\ issued' = issued$

A theorem as given below was added to the generated SAL:

```
th1: theorem State |- G(shelved = set{COPY;}!empty);
```

It says that `shelved` is always empty, which is invalid. It is because `c` of type `COPY` can be added to `shelved` by performing `EnterNewCopy` or `Donate`. Indeed, the SAL model checker reported a counter-example on the verification of this SAL file.

This specification requires a simplification which is applied to the final output, otherwise there will be re-declared state variables. Our system could perform a simple simplification to collapse all state variables and predicates to a reference of the state schema.

As mentioned previously, the first process of a schema composition is renaming which is to rename several state variables to the common names. In this system, the common name is specified to be the same as the name of the state variable, but 0 will be added at the end of this variable. This simplification is achieved by substituting all renamed common variables for their appropriate values obtained from related predicates.

The above example can be translated by Z2SAL. It can also be verified and simulated by the SAL tool.

The following sub-section summarizes results obtained from our experiments with this system. A discussion in these results is also given in this sub-section.

**Summaries of Experiments with the Expansion System** This sub-section discusses several findings found during our experiments with the expansion system. Each finding is discussed in a separate paragraph.

*Re-declaration of State Variables* Re-declaration of state variables is an issue of implementations of renaming and hiding operations. Since a simplification is hard to apply on both operations, these operations cannot be further implemented at the moment.

The current Z2SAL assumes that the first schema definition in a Z specification is a state schema and the second one is an initialization schema. Z2SAL defines also one base module in each SAL specification and accepts only one state schema in each Z specification input, though both SAL and Z allow many modules and state schemas respectively in one specification.

A SAL module specifies a transition system of a finite-state automaton. A Z schema represents a state of a system and a collection of these schemas models behaviour of the system. A state schema is a combination of state variables and predicates of a system.

A restriction on the number of state schemas in a Z specification is also an issue of performing a negation in a schema expansion. Variables and negated predicates in the constructed schema cannot be collapsed into a state schema inclusion. It is because of a problem of re-declared variables. The only way to solve this problem is to define at least two state schemas. The first state schema just defines state variables, whereas the second one defines an inclusion to the first schema and state predicates. However, Z2SAL does not support many state schemas either as discussed earlier.

This restriction affects also how a renaming and a hiding are applied to. Both schema operators cannot be applied to the initialization schema and operational schemas due to the above same problem and instead to the state schema. Furthermore, Z2SAL also enforces us to define the same name for both the constructed schema and the state schema. Thus, the application of these two operators will modify the whole specification.

*The Order of Schema Operators* Another issue in schema expansion is the order or the binding of schema operators, especially when brackets are not added in a definition of schema calculus. Fortunately, operators bindings and associativities can be defined by using built-in options of the BYACC/J parser generator [10]: `left`, `right` and `nonassoc`, which mean left, right and no grouping respectively. Afterwards, several actions can be added in associated grammars to define information about these orders. The order of operators tells us the precedence among them, which is getting higher position, the lower the precedence. Several of these orders are given in the [8].

*Size of Z Specifications* One issue that is important to consider is by having many schema calculus definitions, both a Z specification and a SAL specification are also getting big in sizes. Another important issue is that a size of a SAL specification is roughly twice to four times of its Z specification (see discussion below).

Fortunately, our approach to expand schema calculus definitions scales to larger specifications. However, as the outcomes of our system will be translated by Z2SAL and executed by the SAL tool later, the large specification resulted by our system is possible to be a problem with both tools.

The following describes briefly our experiments with this system. Several tables are presented which summarize these experiments.

Tables 3, 4, and 5 show us results from several examples of our experiments. These examples were obtained from several Z books and they will be discussed below.

Specifications which were used for Experiment 1 to Experiment 8, and Experiment 71 were taken from [2]. It is a library system which has been discussed in Section 3.2.

On the other hand, Experiment 9, Experiment 24 to Experiment 33, Experiment 54 to Experiment 55, and Experiment 58 to Experiment 63 were taken from [22]. This is a simple car park system. The state schema and the initialization schema are as follows:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{CarsPark}} \\
\hline
\mathit{count} : \mathbb{N};\ \mathit{maximum} : \mathbb{N} \\
\hline
\mathit{count} \leq \mathit{maximum} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\textit{InitCarsPark}} \\
\hline
\mathit{CarsPark} \\
\hline
\mathit{count} = 0 \\
\mathit{maximum} = 3 \\
\hline
\end{array}$$

Experiment 10 to Experiment 14, Experiment 23, Experiment 34 to Experiment 44, Experiment 56 to Experiment 57, and Experiment 64 to Experiment 70 were taken from [4]. This system regards with bookings for performances on a concert hall. The state schema and the initialization schema for these experiments are as follows:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{BoxOffice}} \\
\hline
\mathit{seating} : \mathbb{P}\, \mathit{Seat} \\
\mathit{sold} : \mathit{Seat} \nrightarrow \mathit{Customer} \\
\hline
\operatorname{dom} \mathit{sold} \subseteq \mathit{seating} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\textit{InitBoxOffice}} \\
\hline
\mathit{BoxOffice}' \\
\hline
\mathit{sold}' = \varnothing \\
\mathit{seating}' = \mathit{initial\_allocation} \\
\hline
\end{array}$$

Experiment 15 to Experiment 22 were taken from [7].

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Calculator}} \\
\hline
\mathit{store} : \mathit{MEMORY} \rightarrow \mathbb{Z} \\
\mathit{display} : \mathbb{Z} \\
\mathit{arg2} : \mathbb{Z} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Init}} \\
\hline
\mathit{Calculator} \\
\hline
\forall\, m : \mathit{MEMORY} \bullet \mathit{store}(m) = 0 \\
\mathit{display} = 0 \\
\mathit{arg2} = 0 \\
\hline
\end{array}$$

Above are the state and initialization schemas of this specification. This specification is a system of a four function calculator.

**Table 3.** Several Experiments with the Expansion System

| **No** | **Z Specification** | **Details** | **Verification time in secs** | |
|---|---|---|---|---|
| | **(.tex)** | | **Non-simplified** | **Simplified** |
| 1. | `expandingschema_1` | ”$\bigvee$” | 0.063 | 0.031 |
| 2. | `expandingschema_2` | ”$\bigwedge$” | 0.062 | |
| 3. | `expandingschema_3` | ”$\fatsemi$” | 0.03<br>0.733 | |
| 4. | `expandingschema_4` | ”$\bigwedge$”<br>”$\bigvee$, $\bigvee$”<br>”$\bigvee$” | 0.016 | |
| | | | 2.044 | |
| 5. | `expandingschema_5` | ”$\bigwedge$, $\neg$, $\bigwedge$” | 0.031<br>1.654 | |
| 6. | `expandingschema_6` | ”$\bigwedge$, $\big[$, $\big]$” | 0.031<br>0.686 | |
| 7. | `expandingschema_7` | ”$\neg$, $\bigwedge$, $\big[$, $\big]$” | N/A | |
| 8. | `expandingschema_8` | ”$\bigwedge$, $\big[$, $\big]$”<br>”$\neg$, $\bigwedge$, $\big[$, $\big]$”<br>”$\bigvee$” | N/A | |
| 9. | `expandingsch2_4` | ”$\neg$” | N/A | |
| 10. | `expandingsch3_1` | ”$\Rightarrow$” | 0.015 | |
| 11. | `expandingsch3_2` | ”$\bigwedge$, $\Rightarrow$” | 0.032 | |
| 12. | `expandingsch3_4` | ”$\Rightarrow$, $\bigwedge$” | 0.016 | |
| 13. | `expandingsch4_1` | ”$\Leftrightarrow$” | 0.015 | |
| 14. | `expandingsch4_2` | ”$\bigwedge$, $\Leftrightarrow$” | 0.031 | |
| 15. | `expandingsch5_1` | ”$\big[$, /, $\big]$” | N/A | |
| 16. | `expandingsch5_2` | ”$\big[$, /, /, $\big]$” | N/A | |
| 17. | `expandingsch6_1` | ”$\big\backslash$” | N/A | |
| 18. | `expandingsch6_2` | ”$\big\backslash$” | N/A | |
| 19. | `expandingsch7_1` | ”$\fatsemi$” | 0.031 | |
| 20. | `expandingsch8_1` | ”$\forall$” | N/A | |
| 21. | `expandingsch8_2` | ”$\forall$” | N/A | |
| 22. | `expandingsch8_3` | ”$\forall$, $\bigwedge$” | N/A | |
| 23. | `expandingsch8_6` | ”$\exists$” | N/A | |
| 24. | `expandingsch1_1` | ”$\bigwedge$” | 0.0 | |
| 25. | `expandingsch1_2` | ”$\bigwedge$” | 0.016 | |
| 26. | `expandingsch1_3` | ”$\bigwedge$, $\bigwedge$, $\bigwedge$” | 0.0 | |
| 27. | `expandingsch1_4` | ”$\bigwedge$, $\bigwedge$” | 0.015 | |
| 28. | `expandingsch1_5` | ”$\bigvee$, $\bigwedge$” | 0.015 | |
| 29. | `expandingsch1_6` | ”$\bigvee$, $\bigwedge$” | 0.0 | |
| 30. | `expandingsch1_7` | ”$\bigwedge$, $\bigvee$” | 0.015 | |
| 31. | `expandingsch1_8` | ”$\bigwedge$, $\bigvee$” | 0.0 | |

**Table 4.** Several Experiments with the Expansion System (continued)

| No | Z Specification | Details | Verification time in secs | |
|---|---|---|---|---|
| | (.tex) | | Non-simplified | Simplified |
| 32. | `expandingsch1_9` | ”$\wedge$” | 0.0 | |
| 33. | `expandingsch1_10` | ”$\vee$, $\vee$, $\vee$” | 0.0 | |
| 34. | `expandingsch1_11` | ”$\wedge$, $\vee$, $\wedge$” | 0.016 | |
| 35. | `expandingsch1_12` | ”$\wedge$, $\vee$, $\wedge$” | 0.015 | |
| 36. | `expandingsch1_13` | ”$\wedge$, $\vee$, $\wedge$” | 0.016 | |
| 37. | `expandingsch1_14` | ”$\wedge$, $\vee$, $\wedge$” | 0.016 | |
| 38. | `expandingsch1_15` | ”$\wedge$” | 0.016 | |
| 39. | `expandingsch1_16` | ”$\wedge$” | 0.016 | |
| 40. | `expandingsch1_17` | ”$\wedge$” | 0.016 | |
| 41. | `expandingsch1_18` | ”$\wedge$” | 0.031 | |
| 42. | `expandingsch1_19` | ”$\wedge$, $\vee$, $\wedge$” | 0.032 | |
| 43. | `expandingsch1_20` | ”$\wedge$, $\vee$, $\wedge$” | N/A | |
| 44. | `expandingsch1_21` | ”$\wedge$, $\vee$, $\wedge$” | 0.03 | |
| 45. | `expandingsch1_22` | ”$\wedge$, $\vee$” | 0.031 | |
| 46. | `expandingsch1_23` | ”$\wedge$, $\vee$” | 0.032 | |
| 47. | `expandingsch1_24` | ”$\wedge$, $\vee$” | 0.031 | |
| 48. | `expandingsch1_25` | ”$\wedge$, $\vee$” | 0.016 | |
| 49. | `expandingsch1_26` | ”$\wedge$” | 0.015 | |
| 50. | `expandingsch1_27` | ”$\wedge$” | 0.031 | |
| 51. | `expandingsch1_28` | ”$\vee$, $\vee$, $\vee$” | 0.031 | |
| 52. | `expandingsch1_29` | ”$\vee$, $\vee$, $\vee$” | 0.031 | |
| 53. | `expandingsch1_30` | ”$\vee$, $\vee$, $\vee$” | 0.047 | |
| 54. | `expandingsch1_31` | ”$\vee$, $\wedge$, $\vee$” | 0.0 | |
| 55. | `expandingsch1_32` | ”$\vee$, $\vee$, $\wedge$” | 0.015 | |
| 56. | `expandingsch2_1` | ”$\neg$” | N/A | |
| 57. | `expandingsch2_2` | ”$\neg$, $\wedge$” | 0.032 | |
| 58. | `expandingsch2_3` | ”$\neg$” | N/A | |
| 59. | `expandingsch2_5` | ”$\neg$, $\wedge$” | N/A | |
| 60. | `expandingsch2_6` | ”$\neg$, $\wedge$” | N/A | |
| 61. | `expandingsch2_7` | ”$\wedge$, $\neg$” | 0.0 | |
| 62. | `expandingsch2_8` | ”$\wedge$, $\neg$” | 0.0 | |
| 63. | `expandingsch2_9` | ”$\neg$, $\wedge$, $\neg$” | N/A | |
| 64. | `expandingsch3_3` | ”$\wedge$, $\Rightarrow$” | 0.016 | |
| 65. | `expandingsch3_5` | ”$\Rightarrow$, $\wedge$” | 0.015 | |
| 66. | `expandingsch3_6` | ”$\Rightarrow$, $\wedge$” | 0.031 | |
| 67. | `expandingsch3_7` | ”$\Rightarrow$, $\vee$, $\Rightarrow$” | N/A | |
| 68. | `expandingsch3_8` | ”$\wedge$, $\Rightarrow$, $\wedge$” | 0.015 | |
| 69. | `expandingsch3_9` | ”$\wedge$, $\Rightarrow$, $\wedge$, $\Rightarrow$, $\wedge$” | 0.015 | |

**Table 5.** Several Experiments with the Expansion System (continued)

| No | Z Specification | Details | Verification time in secs | |
|---|---|---|---|---|
| | (.tex) | | Non-simplified | Simplified |
| 70. | `expandingsch8_5` | "$\forall$" | N/A | |
| 71. | `expandingschema_9` | "$\bigwedge$, $\neg$, $\bigwedge$"<br>"$\bigwedge$, [, ]"<br>"$\neg$, $\bigwedge$, [, ]"<br>"$\bigvee$"<br>"$\bigwedge$, $\bigvee$" | N/A | |

Experiment 45 to Experiment 53 were taken from [23], but have been modified in several places to be able to be translated by Z2SAL. One of these modifications is to have one state schema. In the original specification, there are references to several different schemas. These references indicate the referenced schemas are state schemas. The modification is necessary to be translated by Z2SAL. A state and initialization schemas are given as follows:

*Flexi*

$Standard\_Hours, Flexitime\_Hours : Time \rightarrow Period$
$worked : Ident \nrightarrow Period; \; in : Ident \nrightarrow Time$

$\mathrm{dom}\, in \subseteq \mathrm{dom}\, worked$

*InitFlexi*

$Flexi$

$in = \varnothing$
$worked = \varnothing$

As can be seen from Table 3, 4, and 5, a simplification has only been performed on an output of `expandingschema_1` specification. It is indicated by two verification times in associated columns. Outputs of `expandingschema_3`, `expandingschema_4`, `expandingschema_5`, and `expandingschema_6` have two verification times in one column. The first time is a verification time with no theorem and the second one is a time with one theorem. There are three specifications that have many schema calculus definitions: `expandingschema_4`, `expandingschema_8`, and `expandingschema_9`. "`N/A`"s in several rows mean that the related specification cannot be translated by Z2SAL. All of these specifications contain re-declarations of state variables. It is because these variables could not be collapsed by our system to references of a state schema.

Regarding size of Z specifications, this will be discussed below. Tables 6, and 7 show us sizes of our Z specifications on this experiments. As can be seen from these three tables, a range of sizes of our Z specification inputs is between 1 and 3 kilobytes, otherwise the ranges are 1 to 8 and 1 to 14 for Z specification outputs and SAL specifications respectively. Sizes of SAL specifications shown in this table are original sizes producing by Z2SAL. As discussed above, a few of these SAL specifications have been modified as required in order to be executed by the SAL tool successfully or have been simplified to their compact form of predicates. Thus, their sizes can be different from the original ones.

"`input`" means a Z specification input file for our system. On the other hand, "`output`" means a Z specification output file generated by our system after performing an expansion process, "N/A"s in `output` means an associated Z specification input could not be expanded by our system either because of errors in the input file or because of bugs on our system, "N/A"s in `SAL specifications` means an associated Z specification could not be translated by Z2SAL. It can also be seen that a "N/A" in `input` makes this Z specification is not possible to be further processed.

## 4 Conclusion and Future Work

Our experiments find that the SAL language is not a case sensitive language. Another finding is that a bug on a Z2SAL translation of a range of numbers is found. This finding convinces us to such a bug since our other experiments with Z2SAL also find this.

All tables, which summarize our experiments, show that majority of our running examples can be redefined or expanded by our system. Several of them can also be translated by Z2SAL, verified by the SAL model checker, or simulated by the SAL simulator.

As a conclusion, redefinition and schema expansion, which pre-process a Z specification, can benefit the scope of translation of Z2SAL. It is because a Z specification can consist of generic constant or schema calculus definitions. This fact can support Z2SAL to translate a variety of Z specifications, which at the end can also support model checking Z specifications.

However, our method of implementing this system seems that our method is not feasible for larger and more complex specifications. It is because such specifications require more time to be translated by Z2SAL and to be executed by the SAL tool.

Expanded schemas can make the larger specification even larger. A more complex generic constant definition means several conditions. It can be more complex types of generic constant variables. It can also be a more complex predicate part of this definition. On the other hand, a more complex schema calculus definition means the definition contains a combination of several schema operators. Inevitably, further work could extend the system so it is able to run more complex Z specifications.

Furthermore, the out of memory error which is often encountered during simulation is also beneficial to be addressed. How abstraction can be applied to the related schemas to reduce the memory consumption is planned to be investigated.

Moreover, re-declaring state or global variables could be approached by implementing a better simplification in predicates. It could also include upgrading Z2SAL to a version that accepts many state schemas and references to them.

Thus, one state schema can be specified to have just a variable part. Another state schema has a predicate part. Having these state schemas, a user can collapse state variables easily without a bother on negated predicates of other state schema.

Other future work is our approach to a SAL translation of a user defined function or constant could also be automated. There are two options for this automation: implementing it as an extension to this system or adding it as an extension to Z2SAL system. It appears that the second option is an easier method to implement.

## Acknowledgment

The author would like to thank John Derrick, Siobhán North and Anthony J.H. Simons for giving the author a chance to work with their Z2SAL, discussions in this tool, and supervision in this study. A lot of thanks are dedicated to ISIHEMORA The Republic of Indonesia for its financial support of this study.

## References

1. Jackson, D.: Abstract model checking of infinite specifications. FME'94: Industrial Benefit of Formal Methods. Springer, 519–531 (1994)
2. Potter, B., Till, D., and Sinclair, J.: An introduction to formal specification and Z. Prentice Hall PTR (1996)
3. West, M.M.: Issues in Validation and Executability of Formal Specifications in the Z Notation. Thesis of University of Leeds (2002)
4. Woodcock, J. and Davies, J.: Using Z: specification, refinement, and proof. Prentice-Hall, Inc. (1996)
5. Derrick, J., North, S., and Simons, A.J.H.: Z2SAL: a translation-based model checker for Z. Formal aspects of computing. Springer, 23 1, 43–71 (2011)
6. Malik, P., Groves, L. and Lenihan, C.: Translating z to alloy. ASM, Alloy, b and Z. Springer, 377–390 (2010)
7. Barden, R., Stepney, S. and Cooper, D.: Z in Practice. Prentice-Hall, Inc. (1995)
8. Spivey, J.M.: The Z notation. Prentice Hall New York (1989)
9. Klein, G.: JFlex - The Fast Scanner for Java. Accessed from http://www.jflex.de/index.html (2015)
10. Hurka, T.: BYACC/J. Accessed from http://byaccj.sourceforge.net (2008)
11. Deitel, H. and Deitel, Paul. J.: Java: How to program. 5th (internatioal) ed. Upper Saddle River, NJ: Prentice-Hall (2003)
12. Siregar, M.U.: Support for Model Checking Z Specifications. IEEE 17th International Conference on Information Reuse and Integration (IRI), 241–248 (2016)
13. Plagge, D. and Leuschel, M.: Validating Z specifications using the ProB animator and model checker. Integrated Formal Methods. Springer, 480–500 (2007)
14. Bolton, C.: Using the alloy analyzer to verify data refinement in Z. Electronic Notes in Theoretical Computer Science. Elsevier, 137 2, 23–44 (2005)
15. De Moura, L., Owre, S. and Shankar, N.: The SAL language manual. Computer Science Laboratory, SRI International, Menlo Park, CA, Tech. Rep. CSL-01-01 (2003)

16. Smith, G. and Wildman, L.: Model checking Z specifications using SAL. ZB 2005: Formal Specification and Development in Z and B. Springer, 85–103 (2005)
17. Derrick, J., North, S., and Simons, A.J.H.: Issues in implementing a model checker for Z. Formal Methods and Software Engineering. Springer, 678–696 (2006)
18. Simons, AJH: The Z2SAL User Guide. Accessed from http://staffwww.dcs.shef.ac.uk/people/A.Simons/z2sal/userguide.html (2012)
19. Bensalem, S., Lakhnech, Y. and Owre, S.: Computing abstractions of infinite state systems compositionally and automatically. Computer Aided Verification. Springer, 319–331 (1998)
20. King, Paul: Printing Z and Object-Z LaTeXdocuments. Department of Computer Science, University of Queensland. (1990)
21. Rann, D. and Turner, J. and Whitworth, J.: Z: a Beginner's Guide. CRC Press (1994)
22. Marris, T.: Z notes (2007)
23. Hayes, I. and Flinn, B.: Specification case studies. Prentice-Hall International London (1987)

**Table 6.** Sizes of Z Specifications

| **Z Specifications** | **Sizes in KB** | | |
|---|---|---|---|
| **(.tex)** | **input** | **output** | **SAL specifications** |
| `expandingschema_1` | 2 | 2 | 7 |
| `expandingschema_2` | 2 | 2 | 6 |
| `expandingschema_3` | 2 | 2 | 6 |
| `expandingschema_4` | 2 | 3 | 10 |
| `expandingschema_5` | 2 | 3 | 9 |
| `expandingschema_6` | 2 | 2 | 4 |
| `expandingschema_7` | 2 | 3 | N/A |
| `expandingschema_8` | 3 | 5 | N/A |
| `expandingsch2_4` | 1 | N/A | N/A |
| `expandingsch3_1` | 1 | 1 | 3 |
| `expandingsch3_2` | 2 | 2 | 4 |
| `expandingsch3_4` | 2 | 2 | 4 |
| `expandingsch4_1` | 2 | 2 | 4 |
| `expandingsch4_2` | 2 | 2 | 5 |
| `expandingsch5_1` | 1 | 1 | N/A |
| `expandingsch5_2` | 1 | 1 | N/A |
| `expandingsch6_1` | 1 | 1 | N/A |
| `expandingsch6_2` | 2 | 2 | N/A |
| `expandingsch7_1` | 1 | 1 | 2 |
| `expandingsch8_1` | 2 | 2 | N/A |
| `expandingsch8_2` | 2 | 2 | N/A |
| `expandingsch8_3` | 2 | 2 | N/A |
| `expandingsch8_6` | 1 | 2 | N/A |
| `expandingsch1_1` | 1 | 1 | 3 |
| `expandingsch1_2` | 1 | 1 | 3 |
| `expandingsch1_3` | 1 | 1 | 3 |
| `expandingsch1_4` | 1 | 1 | 3 |
| `expandingsch1_5` | 1 | 1 | 3 |
| `expandingsch1_6` | 1 | 1 | 3 |
| `expandingsch1_7` | 1 | 1 | 3 |
| `expandingsch1_8` | 1 | 1 | 3 |
| `expandingsch1_9` | 1 | 1 | 3 |
| `expandingsch1_10` | 1 | 1 | 3 |
| `expandingsch1_11` | 1 | 2 | 3 |
| `expandingsch1_12` | 1 | 2 | 4 |
| `expandingsch1_13` | 1 | 2 | 4 |
| `expandingsch1_14` | 1 | 2 | 3 |
| `expandingsch1_15` | 1 | 1 | 3 |
| `expandingsch1_16` | 1 | 1 | 3 |
| `expandingsch1_17` | 1 | 1 | 3 |
| `expandingsch1_18` | 1 | 1 | 3 |
| `expandingsch1_19` | 2 | 2 | 4 |
| `expandingsch1_20` | 2 | N/A | N/A |
| `expandingsch1_21` | 2 | 2 | 4 |

**Table 7.** Sizes of Z Specifications (continued)

| **Z Specification** | **Sizes in KB** | | |
|---|---|---|---|
| **(.tex)** | **input** | **output** | **SAL specifications** |
| expandingsch1_22 | 2 | 2 | 4 |
| expandingsch1_23 | 2 | 2 | 4 |
| expandingsch1_24 | 2 | 2 | 4 |
| expandingsch1_25 | 2 | 2 | 5 |
| expandingsch1_26 | 2 | 3 | 7 |
| expandingsch1_27 | 2 | 3 | 7 |
| expandingsch1_28 | 3 | 5 | 14 |
| expandingsch1_29 | 3 | 5 | 14 |
| expandingsch1_30 | 3 | 5 | 14 |
| expandingsch1_31 | 1 | 1 | 3 |
| expandingsch1_32 | 1 | 1 | 3 |
| expandingsch2_1 | 1 | 1 | N/A |
| expandingsch2_2 | 1 | 1 | 3 |
| expandingsch2_3 | 1 | 2 | N/A |
| expandingsch2_5 | 1 | 2 | N/A |
| expandingsch2_6 | 1 | 2 | N/A |
| expandingsch2_7 | 1 | 1 | 3 |
| expandingsch2_8 | 1 | 1 | 3 |
| expandingsch2_9 | 1 | 2 | N/A |
| expandingsch3_3 | 2 | 2 | 4 |
| expandingsch3_5 | 2 | 2 | 4 |
| expandingsch3_6 | 2 | 2 | 4 |
| expandingsch3_7 | 2 | 2 | N/A |
| expandingsch3_8 | 2 | 2 | 4 |
| expandingsch3_9 | 2 | 2 | 5 |
| expandingsch3_10 | 2 | N/A | N/A |
| expandingsch6_3 | 2 | N/A | N/A |
| expandingsch6_4 | 2 | N/A | N/A |
| expandingsch7_2 | 2 | N/A | N/A |
| expandingsch8_4 | 2 | N/A | N/A |
| expandingsch8_5 | 1 | 2 | N/A |
| expandingschema_9 | 3 | 8 | N/A |